Strategic Port Graph Rewriting for Autonomic Computing

Oana Andrei, Hélène Kirchner

Strategic Port Graph Rewriting for Autonomic Computing

Oana Andrei Hélène Kirchner
INRIA Nancy Grand-Est & LORIA
Campus scientifique BP 239
F-54506 Vandoeuvre-lès-Nancy Cedex, France
First.Last@loria.fr

Abstract

In this paper, we present a high-level formalism based on port graph rewriting, strategic rewriting, and rewriting calculus. We show that this formalism is suitable for modeling autonomic systems and we illustrate its expressivity for modeling properties of such systems using an example of a mail delivery system.

1. Introduction

Autonomic computing [19] refers to self-manageable systems initially provided with some high-level instructions from administrators. This is a concept introduced in 2001 with an intended biological connotation: the simplest biological example is that of the human nervous system where the brain does not need to handle all low-level still vital functions of the body. The four most-important aspects of self-management as presented in [19] are self-configuration, self-optimization, self-healing, and self-protection.

This idea of biologically inspired formalism gained much interest with the recent development of large scale distributed systems such as service infrastructures and Grids. For such systems, there is a crucial need for theories and formal frameworks to model computations, to define languages for programming and to establish foundations for verifying important properties of these systems. Several approaches contributed to this ambitious goal. Without exhaustivity, let us mention in particular the brane calculus [11, 17] and the bigraphical reactive systems [27], but also several calculi inspired from biology such as [29, 12, 25].

Another connected approach is provided by chemical programming which uses the chemical reaction metaphor to express the coordination of computations. This metaphor describes computation in terms of a chemical solution in which molecules (representing data) interact freely according to reaction rules. Chemical solutions are represented by multisets (data-structure that allows several occurrences of the same element). Computation proceeds by rewritings, which consume and produce new elements according to conditions and transformation rules. The Gamma formalism was first proposed in [7] and later extended to HOCL in [5, 6] for modeling self-organizing and autonomic systems or grids in particular.

Beyond the chemical programming idea, another approach presented in [13], called the Organic Grid, is similarly a radical departure from current approaches and is inspired by the self-organization property of complex biological systems.

Our previous work on biochemical applications led us to consider the structure of port graph to model interactions between molecules or proteins [2, 1].

In this paper, we propose port graphs as a formal model for distributed resources and grid infrastructures. Each resource is modeled by a node with explicit connection points called ports. We model the lack of global information, the autonomous and distributed behavior of components by a multiset of port graphs and rewrite rules which are applied locally, concurrently, and non-deterministically. Hence the computations take place wherever it is possible and in parallel. This approach also provides a formal framework to reason about computations and to verify desirable properties. Moreover strategic port graph rewriting takes into account control on computations by allowing to chain rewrite rules. By lifting port graph rewriting to a calculus, we are able to express rules and strategies as port graphs and so to rewrite them as well. The calculus also permits the design of rules that create new rules. In addition, our graph-based formalism is visual, so hopefully easy to understand. We borrow various concepts from graph theory, in particular from graph transformations [16], and different representations for graphs already intensively formalised, like the set-based or the categorical approaches.

The paper is structured as follows. Section 2 introduces the port graphs, while Section 3 introduces the port graph

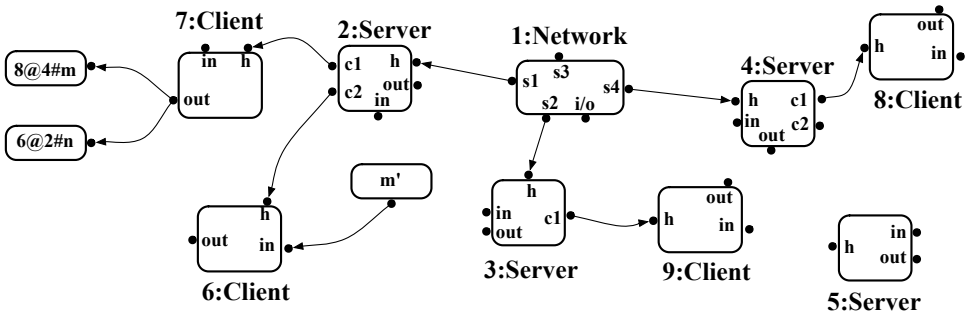

Figure 1. A mail system configuration

rewrite rules as port graphs and defines the matching and the rewriting relation for port graphs. Section 4 provides a presentation of abstract strategies in the context of abstract reduction systems, which can be easily adjusted for port graph rewriting system to obtain a definition of the strategic port graph rewriting. Having all ingredients, in Section 5 we give the main ideas of a high-level calculus for port graphs, and in Section 6 we show that this calculus is a suitable formalism for modeling autonomic systems. We illustrate the concepts and ideas presented in this paper with examples from a mail delivery system example borrowed from [8]. In Section 7 we give some suggestions on expressing properties of a modeled system as strategies placed at the same level as the specification of the system.

2. Port Graphs

We refine the notion of multigraphs, which are graphs with multiple edges and loops, by adding explicit connection points, called *ports*, to nodes; then edges attach, more specifically, to ports of nodes. We represent graphically a node as a box with the identifier and the name placed outside the box and a port as a small point on the surface of the box.

Example 1 *In order to illustrate our approach and the proposed concepts, we develop the example of a mail delivery system borrowed from [8]. It consists of a network of several mail servers each with its own address domain; the clients send messages for other clients first to their server domain, which in turn forwards them to the network and recovers the messages sent to its clients. Servers are distributed resources with connections between them, when sending and receiving the messages.*

In Fig. 1 we illustrate an initial configuration of the mail delivery system. The network is a node with several ports, each port being connected to at most one server. A server node has a handler port for connecting to the network, and several ports for the clients. A client node has a handler port for connecting to a server. All client, server and network nodes have two ports for the incoming and outgoing messages respectively. Messages are nodes with only one port and their names have the form (rec @ $domain$ # m) where rec is the identifier of the recipient client, $domain$ is the identifier of the server domain, and m the body of the message. If redundant, the domain and/or the client identifiers are removed (when arrived in the server domain or at the client). In the system, the server identified by **5** *is disconnected from the network node, hence it is crashed.*

Let $\mathscr{N}$ be a finite set of node names and $\mathscr{P}$ be a finite set of port names. A p-signature $\nabla = \langle \mathscr{N}, \mathscr{P} \rangle$ is a function associating to each node name from $\mathscr{N}$ a set of port names from $\mathscr{P}$. Hence ∇ is a set of pairs of node name and set of port names.

Definition 1 (Port graph) *A* port-graph *over a p-signature $\nabla = \langle \mathscr{N}, \mathscr{P} \rangle$ takes the form $G = (V, E, \iota, s, t, l)$ where:*

- *V is a finite set of nodes (also referred to as node identifiers);*
- *$\iota : V \to \nabla$ assigns a name and a port set to each node, with $\iota(v) = (n, P)$;*
- *$E \subseteq \{(v^\frown p, u^\frown r) \in (V \times \mathscr{P})^2 \mid p \in \nabla(v), r \in \nabla(u)\}$ a finite multiset of edges;*
- *$s, t : E \to V \times \mathscr{P}$ the usual source and target functions;*
- *$l = (l_V, l_E)$ is the labeling function associating to each node $v \in V$ the triple consisting of the identifier, the name, and the port set, $l_V(v) = \langle v : n \parallel P \rangle$ where $\iota(v) = (n, P)$, and to each edge $(v^\frown p, u^\frown r) \in E$ the couple formed by the source port and the target port, $l_E((v^\frown p, u^\frown r)) = (p, r)$.*

Note that, by definition, the ports associated to a node are pairwise distinct.

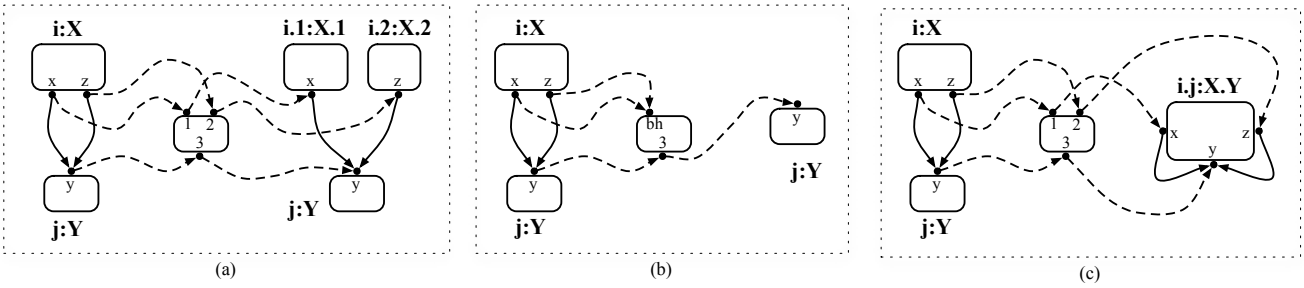

Figure 2. Some port graph rewrite rules: (a) splitting the node i in two; (b) deleting the node i; (c) merging the nodes i and j.

Let $\mathscr{P} = \{a, b, c, \ldots\}$ and $\mathscr{N} = \{A, B, C, \ldots\}$ the sets of constants denoting ports and names respectively. We consider variables ports and names as well, denoted by $\mathcal{X}_{\mathscr{P}} = \{x, y, z, \ldots\}$ and $\mathcal{X}_{\mathscr{N}} = \{X, Y, Z, \ldots\}$ respectively. We represent the node identifiers by non-empty sequences of integers. We denote by $\mathcal{V}ar(G)$ the set of variables occurring in G.

3. Matching and Rewriting Port Graphs

The evolution of the distributed system is modeled via port graph transformations, themselves expressed by port graph rewrite rules and the generated rewriting relation. To support intuition, in the mail system example, rules express what happens when a client sends a mail to a client in the same network as we will show in Example 2.

3.1. Port Graph Rewrite Rule

Definition 2 (Port graph rewrite rule) *A* port graph rewrite rule $L \Rightarrow R$ *is a port-graph consisting of:*

- *two port-graphs L and R over the p-signature $\langle \mathscr{N} \cup \mathcal{X}_{\mathscr{N}}, \mathscr{P} \cup \mathcal{X}_{\mathscr{P}} \rangle$ called, as usual, the left- and right-hand side respectively, such that all node identifiers are variables,*
- *one special "arrow" node, $\Rightarrow$ that has a port for each port in L which is preserved in R, and the* black hole *port, named bh,*
- *edges from L to $\Rightarrow$ mapping each port in L uniquely to a port of $\Rightarrow$ if it is not deleted, and to the black hole port otherwise,*
- *edges from $\Rightarrow$ to R, mapping each port different from the black hole to a port in R.*

A port-graph rewrite systems *is a finite set of port-graph rewrite rules.*

The arrow node together with its adjacent edges embed the correspondence between elements of L and elements of R. We illustrate some port graph rewrite rules in Fig. 2.

We call a port graph that does not contain the arrow node, a *concrete port graph*. In particular, concrete port graphs correspond to configurations or patterns in a modeled system.

Example 2 *We illustrate in Fig. 3 the basic rules for the mail system. Since the correspondence between the left- and right-hand sides of the rules are the identities, we simplify the graphs by not detailing the arrow node. A mail sent by a client goes to its server: if the mail is sent to a client in the same server domain then it goes to the input port by* **r1***, otherwise to the outgoing port by* **r2***. By rule* **r3** *a server forwards a mail to a client if he is the recipient. Rule* **r4** *specifies that a server forwards a mail to the network if its recipient is not in the domain, while rule* **r5** *specifies that the network forwards a mail to the appropriate server according to the server domain information contained in the mail.*

3.2. Matching

A *port graph morphism* assigns the nodes, ports, and edges of a given graph to the nodes, ports, and edges of another graph while preserving adjacency and the membership of port to nodes.

We say that the left-hand side L of a port graph rewrite rule *matches* a port graph G if there is a triple of the form $(g, G^-, \mathcal{B})$ where:

- g is a port-graph morphism such that $g(L)$ is a subgraph of G,
- G^- is the *context graph*, i.e., $G^- = G \setminus g(L)$, given by the set of nodes $V_{G^-} = V_G \setminus V_{g(L)}$, and the set of edges $E_{G^-} = \{(u^\frown p, v^\frown r) \in E_G \mid u, v \in V_{G^-}\}$.

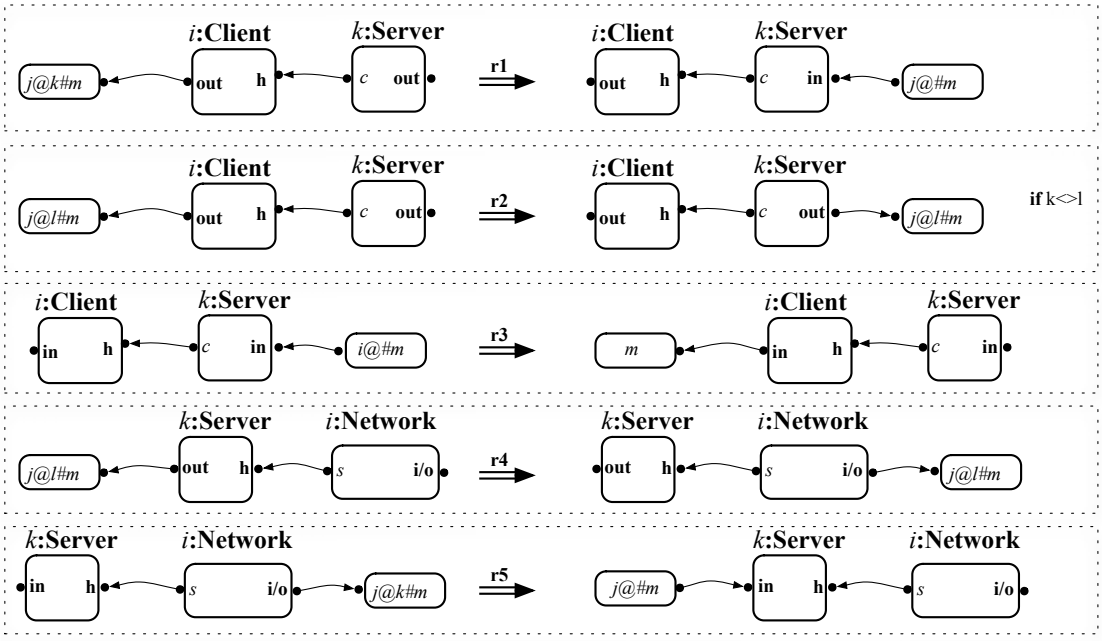

Figure 3. Basic rules for the mail delivery system

- $\mathcal{B}$ is the set of *bridges*, where a bridge is an edge with one endpoint in the context graph and the other in the matched subgraph.

Such a triple gives a decomposition of G as $G^- \cup g(L) \cup \mathcal{B}$ which we write as $G = G^-[g(L)]_{\mathcal{B}}$.

In general, several solutions are provided for the matching problem, since there are several possibilities to choose match nodes and bridges. Intuitively this is because port graph matching involves list pattern matching on ports. One may decide to choose one solution according to some criterion, for instance the first solution. Of course, another choice may lead to another result.

An anti-pattern matching algorithm for graphs can be developed based in the anti-pattern matching on algebraic terms [22]. This allows us to use negative application conditions in graph rewrite rules, like the non-existence of certain nodes, edges, or subgraphs [18].

3.3. Rewriting

Let $L \Rightarrow R$ be a port graph rewrite rule and G a concrete port graph such that there is a matching morphism g for L in G. The delicate point of applying $L \Rightarrow R$ to G is to properly define the replacement of $g(L)$ by $g(R)$ in G and the way $g(R)$ is reconnected with G.

The application of the port graph rewrite rule $L \Rightarrow R$ to G can be decomposed in the following steps as illustrated in Fig. 4:

(i) find the matching morphism g, the context G^-, and the bridges $\mathcal{B}$ (their orientation is not important in the drawing);
(ii) identify the left-hand side of the rule with the isomorphic subgraph in G;
(iii) translate the bridges $\mathcal{B}$ following the instantiated arrow node to obtain $\mathcal{B}_g$;
(iv) remove the arrow node and the isomorphic subgraph $g(L)$ which is now disconnected from the context graph.

If a bridge has an endpoint in $g(L)$ that is changed by the rewrite rule, then it cannot be simply reconnected to $g(R)$ during the replacement. However, the arrow node $\Rightarrow$ together with the incident edges trace the change of this endpoint, providing the way of reconnecting which we call the arrow-translation.

Definition 3 (Port graph rewriting) *Given a port-graph rewrite system $\mathcal{R}$, an concrete port graph G* rewrites *to a port-graph G', denoted by $G \rightarrow_{\mathcal{R}} G'$, if there exists a port-graph rewrite rule $L \Rightarrow R$ in $\mathcal{R}$ and a matching $(g, G^-, \mathcal{B})$ of L against G such that $G = G^-[g(L)]_{\mathcal{B}}$ and $G' = G^-[g(R)]_{\mathcal{B}_g}$. In order to emphasize the used rewrite rule, a rewrite step may also be denoted $G \rightarrow_{L \Rightarrow R} G'$.*

Of course, if a condition or a constraint is associated to the rewrite rule $L \Rightarrow R$, its satisfaction is required for applying the rewriting of G to G'.

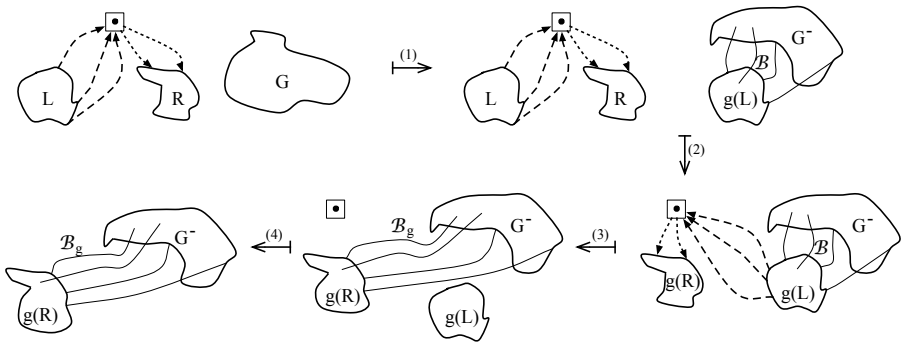

Figure 4. Rewriting steps

4. Strategic Port Graph Rewriting

The port graph transformation of Definition 3 is an instance of the general framework of abstract reduction systems [30, 23, 24].

Definition 4 (Abstract reduction system) *An* abstract reduction system *(ARS) is a labelled oriented graph* $(\mathcal{O}, \mathcal{S})$. *The nodes in* $\mathcal{O}$ *are called* objects*, the oriented edges in* $\mathcal{S}$ *are called* steps*.*

Here an object of $\mathcal{O}$ is a set of port graphs and a step in $\mathcal{S}$ is a port graph rewriting step.

Definition 5 (Derivation) *Let* $\mathcal{A}$ *be a given ARS* $\mathcal{A}$*:*

1. *A* reduction step *is a labelled edge* ϕ *together with its source* a *and target* b*. This is written* $a \rightarrowtriangle^{\phi}_{\mathcal{A}} b$*, or simply* $a \rightarrowtriangle^{\phi} b$ *when unambiguous.*

2. *A* $\mathcal{A}$-derivation *is a path* π *in the graph* $\mathcal{A}$*.*

3. *When it is finite,* π *can be written* $a_0 \rightarrowtriangle^{\phi_0} a_1 \rightarrowtriangle^{\phi_1} a_2 \ldots \rightarrowtriangle^{\phi_{n-1}} a_n$ *and we say that* a_0 *reduces to* a_n *by the derivation* $\pi = \phi_0\phi_1 \ldots \phi_{n-1}$*; this is also denoted by* $a_0 \rightarrowtriangle^{\pi} a_n$*. The* source *of* π *is the object* a_0 *and its domain is defined as the singleton* $dom(\pi) = \{a_0\}$*. The* target *of* π *is the object* a_n *and the application of the derivation* π *to* a_0 *is the singleton denoted* $[\pi](a_0) = \{a_n\}$*.*

4. *A derivation is* empty *when its source and target are the same. The empty derivation issued from* a *is denoted by* id_a*.*

5. *The* concatenation *of two derivations* π_1 *and* π_2 *is defined when* $dom(\pi_1) = \{a\}$ *and* $[\pi_1](a) = dom(\pi_2)$*. Then* $\pi_1;\pi_2$ *denotes the new* $\mathcal{A}$*-derivation* $a \rightarrowtriangle^{\pi_1}_{\mathcal{A}} b \rightarrowtriangle^{\pi_2}_{\mathcal{A}} c$ *and* $[\pi_1;\pi_2](a) = [\pi_2]([\pi_1](a)) = \{c\}$*.*

Note that an $\mathcal{A}$-derivation is the concatenation of its reduction steps. The notion of abstract strategy is introduced in [20] and applies to port graph rewriting.

Definition 6 (Abstract strategy) *For a given ARS* $\mathcal{A}$*:*

1. *An* abstract strategy ζ *is a subset of the set of all derivations (finite or not) of* $\mathcal{A}$*.*

2. *Applying the strategy* ζ *on an object* a*, denoted by* $[\zeta](a)$*, consists of the set of all objects that can be reached from* a *using a derivation in* ζ*:*
$[\zeta](a) = \{b \mid \exists \pi \in \zeta \text{ such that } a \rightarrowtriangle^{\pi} b\} = \{[\pi](a) \mid \pi \in \zeta\}$.
When no derivation in ζ *has for source* a*, we say that the strategy application on* a *fails.*

3. *Applying the strategy* ζ *on a set of objects consists in applying* ζ *to each element* a *of the set. The result is the union of* $[\zeta](a)$ *for all* a*.*

4. *The* domain *of a strategy is the set of objects that are source of a derivation in* ζ*:*

$$dom(\zeta) = \bigcup_{\delta \in \zeta} dom(\delta)$$

5. *The strategy that contains all empty derivations is* $Id = \{id_a \mid a \in \mathcal{O}\}$*.*

Note that, following from the previous definition, a strategy is not defined on all objects of the ARS, hence it is a partial function. A strategy that contains only infinite derivations from a source $\{a\}$ applies to the object a and returns the empty set. The empty set of derivations is a strategy called $Fail$; its application always fails.

We can formalize strategic port graph rewriting in this context:

Definition 7 (Strategic port graph rewriting) *Given an abstract reduction system* $\mathcal{A} = (\mathcal{O}_{\mathcal{R}}, \mathcal{S}_{\mathcal{R}})$ *generated by*

a port graph rewrite system $\mathcal{R}$, and a strategy ζ of $\mathcal{A}$, a strategic port graph rewriting derivation *(or port graph rewriting derivation under strategy ζ) is an element of ζ. A* strategic port graph rewriting step *under ζ is a rewriting step $G \rightarrow_{\mathcal{R}} G'$ that occurs in a derivation of ζ. This is also denoted $G \rightarrow_{\zeta} G'$.*

The formalisation of abstract reduction system and abstract strategies allows then to define properties like termination (all relevant derivations are of finite length) and confluence (all relevant derivations lead to the same object). Their precise definitions can be found in [20]. Termination and confluence for Port Graph Rewriting can be based on criteria developed for graphs like the one in [28]. Similar sufficient conditions for strategic port graph rewriting are yet to be explored.

A strategy can be described by enumerating all its elements or more suitably by a *strategy language*. Various approaches have been followed, yielding different strategy languages such as ELAN [21, 10], Stratego [31], TOM[1] [3, 4] or more recently Maude [26]. All these languages share the concern to provide abstract ways to express control of rule applications, by using reflexivity and the meta-level for Maude, or the notion of rewriting strategies for ELAN or Stratego. Strategies such as bottom-up, top-down or leftmost-innermost are higher-order features that describe how rewrite rules should be applied. TOM, ELAN and Stratego provide flexible and expressive strategy languages where high-level strategies are defined by combining low level primitives. We describe below the main elements of the TOM strategy language that are of interest for the example developed in this paper.

Following [20], we can distinguish two classes of constructs in the strategy language: the first class allows construction of derivations from the basic elements, namely the rewrite rules. The second class corresponds to constructs that express the control, especially left-biased choice (or first). Moreover, the capability of expressing recursion in the language brings even more expressive power.

Elementary strategies. An elementary strategy is either $Identity$ which corresponds to the set Id of all empty derivations, $Fail$ which denotes the empty set of derivations $Fail$, or a set of rewrite rules $\mathcal{R}$ which represents one-step derivations with rules in $\mathcal{R}$ at the root position. $Sequence(\zeta_1, \zeta_2)$, also denoted by $\zeta_2; \zeta_1$, is the concatenation of ζ_1 and ζ_2 whenever it exists: for a given port graph G, $[\zeta_1; \zeta_2](G) = [\zeta_2]([\zeta_1](G))$.

Control strategies. A few constructions are needed to build derivations, branching and to take into account the structure of the objects.

first $First(\zeta_1, \zeta_2)$ applies the first strategy if it does not fail, otherwise it applies the second strategy; it fails if both strategies fail:
$[First(\zeta_1, \zeta_2)](G) = [\zeta_1](G)$ if $[\zeta_1](G)$ does not fail else $[\zeta_2](G)$.

fixpoint The μ recursion operator (comparable to rec in OCaml) is introduced to allow the recursive definition of strategies. $\mu x.\zeta$ applies the derivation in ζ with the variable x instantiated to $\mu x.\zeta$, i.e., $\mu x.\zeta = \zeta[x \leftarrow \mu x.\zeta]$

All these strategies are then composed to build other useful strategies. A composed strategy is for instance $Try(\zeta) = First(\zeta, Id)$ which applies ζ if it can, and performs the identity strategy Id otherwise. Similarly, the $Repeat$ combinator is used in combination with the fixpoint operator to iterate the application of a strategy: $Repeat(\zeta) = \mu x.Try(\zeta; x)$.

5 A Calculus of Port Graphs

We now have all ingredients for defining the rewriting calculus of port graphs (ρ_{pg}-calculus). The ρ_{pg}-calculus generalizes ρ-calculus [14] and ρ_g-calculus [9]. It inherits from ρ-calculus the fact that it generalizes λ-calculus through a much greater abstraction power that considers for matching not only a variable like in λ-calculus but a port graph with variables.

5.1. Syntax

We give the syntax of the calculus in Fig. 5. The objects of the calculus are port graphs that can be either a variable, a concrete port graph, a port graph rewrite rule (also called abstraction), the application of an abstraction to a port graph, or a multiset of such entities. The formal definition of the concrete port graphs in $\mathcal{P}$ and examples are presented in Sect. 2.

Port Graphs			
$\mathcal{G}$	::=	$\mathcal{X}$	(Variables)
	|	$\mathcal{P}$	(Concrete port graphs)
	|	$\mathcal{A}$	(Abstraction)
	|	$\mathcal{A}@\mathcal{G}$	(Application)
	|	$\mathcal{G}\ \mathcal{G}$	(Juxtaposition)
Abstractions			
$\mathcal{A}$	::=	$\mathcal{P} \Rightarrow \mathcal{P}$	(Port graph rewrite rule)

Figure 5. Syntax of the ρ_{pg}-calculus

The juxtaposition operator for constructing multisets of port graphs is associative and commutative, i.e., $(G_1\ G_2)\ G_3 = G_1\ (G_2\ G_3)$ and $G_1\ G_2 = G_2\ G_1$, for any $G_1, G_2, G_3 \in \mathcal{G}$. In the same time, the juxtaposition

[1] http://tom.loria.fr

operator models a random and concurrent choice of an abstraction A and a concrete port graph G such that the application of A on G is successful. The application operator is a particular node with ports, denoted @, and it models the effective interaction between an abstraction and a port graph by connecting them.

5.2. Basic Reduction Semantics

The semantics of the ρ_{pg}-calculus corresponds to a system whose states consist of a multiset of concrete port graphs and port graph rewrite rules where any two entities interact non-deterministically as in a Brownian motion. Two juxtaposed port graphs are not reduced in the calculus, whereas the juxtaposition of an abstraction and a port graph gives rise to an application via the following rule:

$$A\ G \longmapsto A@G \tag{1}$$

The application operator @ is defined to apply an abstraction A to a concrete port graph G. All steps computing $A@G$, including matching and replacing, are expressible using port graphs by considering additional auxiliary nodes and extending the reduction relation with some graphical reduction rules. This mechanism is internalized in the calculus and we do not present it here since it is too technical for the purpose of this paper. A successful reduction of $A@G$ yields a port graph G' such that $G \rightarrow_A G'$ according to a chosen matching solution, while a matching failure returns the initial abstraction and port graph unchanged:

$$A@G \longmapsto G' \quad \textbf{if } G \rightarrow_A G' \tag{2}$$

$$A@G \longmapsto A\ G \quad \textbf{otherwise} \tag{3}$$

Instead of having this highly non-deterministic and non-terminating behaviour of port graph rewrite rules application, one may want to introduce some control to compose or choose the rules to apply, possibly exploiting failure information. This is possible by defining strategies as port graph rewrite rules, and by introducing an explicit port graph reduced to a failure node.

In the following, when we want to emphasize the use of strategies as abstractions, the reduction rule (2) giving the basic semantics of the juxtaposition operator is written by replacing the abstraction A by a strategy S.

5.3. Failure Handling

A successful computation of $S@G$ yields a concrete port graph G' such that G rewrites to G' by strategic port graph rewriting under the strategy S. If a failure occurs during the matching reduction rules, we handle it explicitly by considering a failure node, denoted by stk:

$$S@G \longmapsto G' \quad \textbf{if } G \rightarrow_S G' \tag{4}$$

$$S@G \longmapsto \mathsf{stk} \quad \textbf{otherwise} \tag{5}$$

In addition, the following two reductions concerning stk are necessary for defining the basic cases of the definitions of other strategies (like `first` and `seq` later on):

$$S@\mathsf{stk} \longmapsto \mathsf{stk} \tag{6}$$

$$S\ \mathsf{stk} \longmapsto S \tag{7}$$

5.4. Strategies as Abstractions

In this section, we define strategies as objects of the calculus, using the basic constructs, as one can do in λ-calculus or γ-calculus. For such definition we use a similar approach to the one used in [15] where rewrite strategies are encoded by rewrite rules. Let us consider for the rewrite strategies given in Sect. 4 the following objects: id for Id, `fail` for $Fail$, `first` for $First$, `seq` for $\_;\_$, `try` for Try, `repeat` for $Repeat$. The application operator for strategies $[\_](\_)$ from Sect. 4 corresponds to the application operator @. We also need to extend the syntax of the calculus by considering abstractions with arbitrary port graphs in the right-hand side, i.e., of the form $\mathcal{P} \Rightarrow \mathcal{G}$.

Let S, S_1, S_2 denote strategies. We encode the strategies as the following port graph rewrite rules:

$$\begin{aligned}
\mathsf{id} &\triangleq X \Rightarrow X \\
\mathtt{fail} &\triangleq X \Rightarrow \mathsf{stk} \\
\mathtt{seq}(S_1, S_2) &\triangleq X \Rightarrow S_2@(S_1@X) \\
\mathtt{first}(S_1, S_2) &\triangleq \\
&\quad X \Rightarrow ((S_1@X)\ \ (\mathsf{stk} \Rightarrow S_2@X)@(S_1@X))
\end{aligned}$$

The two composed strategies Try and $Repeat$ are then easily defined as follows:

$$\begin{aligned}
\mathtt{try}(S) &\triangleq \mathtt{first}(S, \mathsf{id}) \\
\mathtt{repeat}(S) &\triangleq \mathtt{try}(\mathtt{seq}(S, \mathtt{repeat}(S)))
\end{aligned}$$

5.5. Refined Reduction Semantics

Building now on these strategy definitions, we can restate properly the main reduction rule modeling the interaction between two port graphs using a failure catching mechanism: if $S@G$ reduces to the failure, i.e., to the stk node, then the strategy $\mathtt{try}(\mathsf{stk} \Rightarrow S\ G)$ restores the initial port graphs subjects to reduction.

$$S\ G \longmapsto \mathtt{try}(\mathsf{stk} \Rightarrow S\ G)@(S@G) \tag{8}$$

Due to the previous definitions of `try` and `first`, and assuming that S is itself built from basic rewrite rules and from the constructions of the strategy language, the right-hand side of rule (8) is (or reduces to) a port graph expressed in the syntax defined in Figure 5 enriched with stk.

Two further refinements are worth considering, in order to handle multiple results and to ensure, whenever needed, persistency of abstractions.

5.6. Multiple results

The successful application of an abstraction or strategy to a concrete port graph produces a new concrete port graph, built according to one chosen matching solution. An alternative would be to consider a structure of all concrete port graphs corresponding to the different matching solutions.

For instance, we may consider the set structure with stk neutral element, and $\_ \wr \_$ the associative and commutative constructor. Then an interaction $S@G$ reduces either to stk or to a structure $G_1 \wr \ldots \wr G_k$ with $G_i \neq \mathsf{stk}$, for every $i = 1, \ldots, k$ and the following reduction rule is necessary for defining the interaction of a strategy with such a structure:

$$S\ (G_1 \wr G_2) \longmapsto (S\ G_1) \wr (S\ G_2) \tag{9}$$

However we do not develop further this alternative in this paper.

5.7. Persistent Strategies

At this level of definition of the calculus, rules or strategies are consumed by a non-failing interaction with a concrete port graph. One advantage is that, since we work with multisets of port graphs, a rule or a strategy can be given a multiplicity, and each interaction between the rule or strategy and another port graph consumes one occurrence. This permits controlling the maximum number of times an interaction can take place.

But sometimes, it may be suitable to have persistency of the information concerning the available abstraction and thus the persistency of a given possible interaction. In this case, the abstraction should not be consumed by the reduction. For that purpose, we define the *persistent* strategy that applies a strategy given as argument and, if successful, replicates itself. Indeed a failure will remain a failure: $S! \triangleq X \Rightarrow \mathtt{first}(\mathsf{stk} \Rightarrow \mathsf{stk}, Y \Rightarrow Y\ S!)@(S@X)$

6. Strategies for Autonomic Computing

In autonomic computing, systems and their components reconfigure themselves automatically according to directives (rewrite rules and strategies) given initially by administrators. Based on these primary directives and their acquired knowledge along the execution, the systems and their components seek new ways of optimizing their performance and efficiency *via* new rewrite rules and strategies that they deduce and include in their own behavior. Since there is no ideal system, functioning problems and malicious attacks or failure cascades may occur, and the systems must be prepared to face them and solve them. Let us consider here four aspects that an autonomic system must handle, namely self-configuration, self-healing, self-protection and self-optimization. In the following, we show how these aspects can be handled by an autonomic mail delivery system, inspired from [8] and modeled using the ρ_{pg}-calculus. For this purpose some additional rewrite rules are defined. In addition to the previously presented concepts, we assume a few more information available in the nodes and ports, which corresponds to existing notions in graph theory. In particular, each port has a degree information that counts the number of incoming and outgoing edges connecting it to other nodes of the concrete port graph. In our running example, this information allows us to express, through conditions in the rules, that a port is saturated, or on the contrary that it has no incident edge.

6.1. Self-configuration

The self-configuration is simply described by the concurrent application of the five rules given in Fig. 3 using the reduction semantics introduced in Sect. 5.

Rule **r6** in Fig. 6 specifies that a client recipient of a mail telling him to migrate to a server k' does so if the server k' has a free client port. We model a free client port on the server k' by a condition on its degree, specifying that there is no incident edge to that port. In a more visual way, we pictured this condition with a slashed edge. But how to deal with mails that will probably arrive later on the server k for the client i? The problem is solved by introducing in the system a new rule that will update the address of the migrating clients. It is possible that the application of a rewrite rule on a port graph introduces, besides modifying the port graph, a new rewrite rule. In addition, before a client migrates, he must get all mails addressed to him that are already on the server; this is modeled via the strategy $\mathtt{first}(\mathtt{repeat}(\mathbf{r3}), \mathbf{r6})$.

Another interesting problem may concern the operations of splitting and merging servers. Then biologically inspired port graph rules from Fig. 2 (a) and (c) could be applied as well for servers.

6.2. Self-healing

An autonomic system detects when a server crashes and the connection of the crashed server to the network is cut. It is expected to repair the problem of the clients connected to the crashed server and of the mails that were about to be sent from that particular server. Rule **r7** creates a temporary server named **TServer** as a copy of the crashed server and connects it to the network (assuming there is a procedure checking if a server failed). The arrow node of the rule encodes the correspondence of all ports of the server node k, and consequently, all connections with the crashed server are recovered by the temporary server. Note that, for simplifying the graphical representation, we do not include all

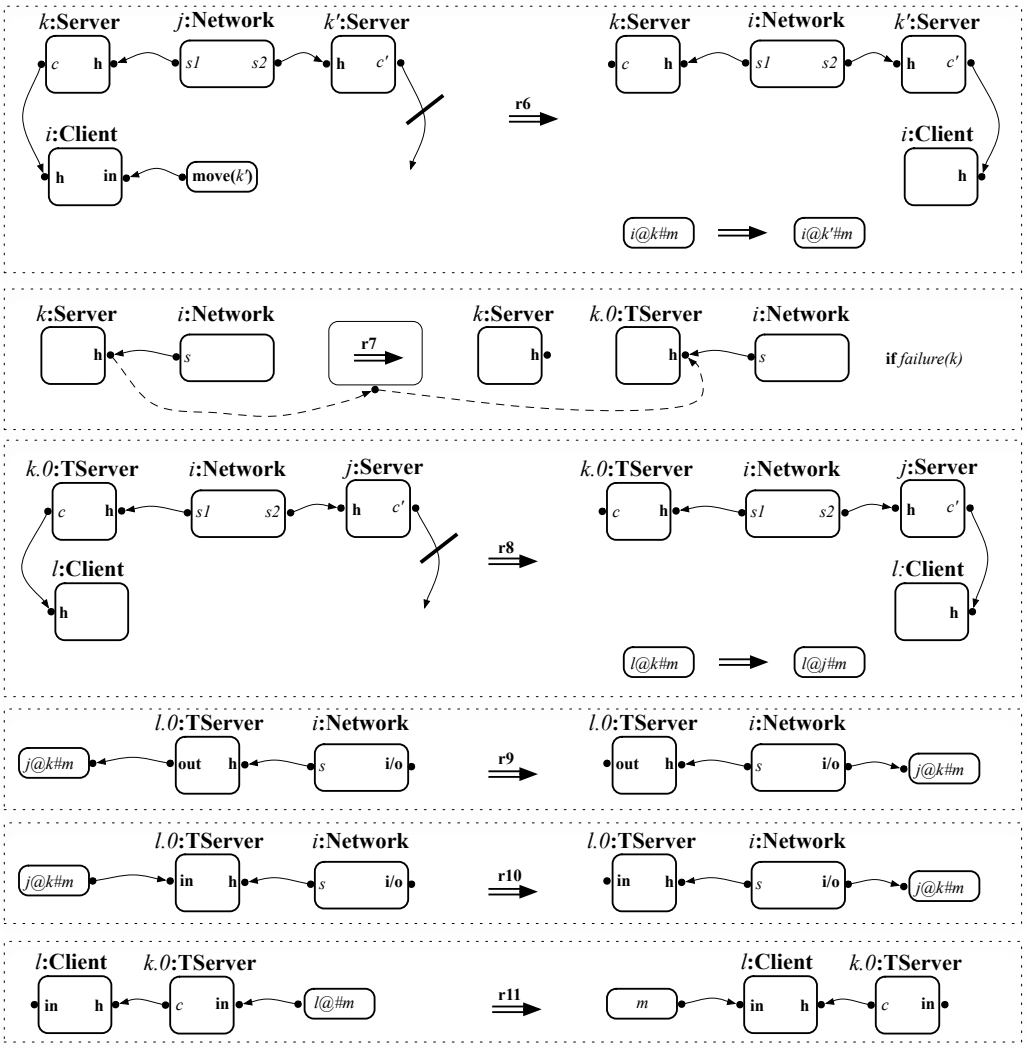

Figure 6. Rules for self-configuration and self-healing in the mail delivery system

edges incident to the arrow node, but just the relevant ones.

Since the server replacing the crashed one is temporary, all the clients must try rapidly to connect to not fully occupied servers in the system (rule **r8**). We graphically express that the server node j has a free client port, i.e., with no incident edges. The mails are forwarded to the network node via the rules **r9** and **r10**. But all this must not happen before sending the mails already there to the clients of the crashed server (rule **r11**). Instead of simply adding all these rules to the system, we add the strategy $\mathbf{r7}; \texttt{repeat}(\mathbf{r11}); \texttt{repeat}(\texttt{first}(\mathbf{r8}, \mathbf{r9}, \mathbf{r10}))$. By composing these rules in a strategy, the recovery from the server crash is assured. A rule should delete the temporary server when it no longer has clients nor pending messages.

6.3. Self-protection

When a spam arrives at a server node, the filtering rule **r12** deletes it, assuming that the server has a procedure for deciding when a mail is a spam. The rules **r13** and **r14** are analogous to **r12** but for a client node and a network node, assuming as well that both entities have their own spam detection procedure. In order to limit spam sending, the rule **r14** should have a higher priority than **r5**, and the rule **r12** a higher priority than **r3**. Then we replace **r3** and **r12** by $\texttt{try}(\mathbf{r12}); \mathbf{r3}$, and **r5** and **r13** by $\texttt{try}(\mathbf{r14}); \mathbf{r5}$.

When a client receives a mail and, based on a spam decision procedure, concludes that the mail is a spam, it deletes the mail and provides the server with a new rule specifying that from now on the server node should delete all mails of this kind. This behavior is specified by the rule **r15** in

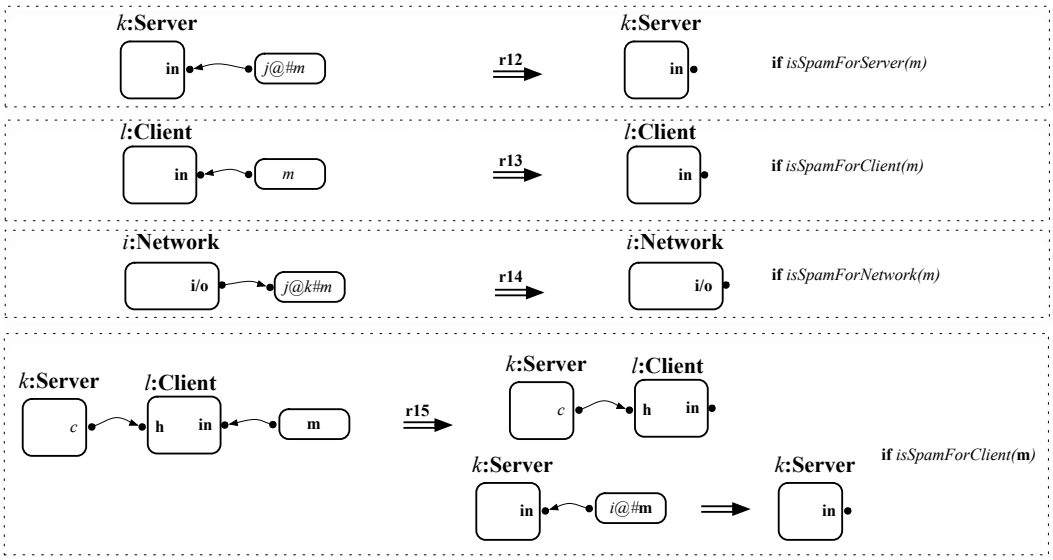

Figure 7. Rules for self-protection in the mail delivery system

Fig. 7.

6.4. Self-optimization

Assuming that a server can determine when it is saturated, i.e., when it reaches the maximal load of incoming messages, a particular type of server for equilibrating the load of the saturated server is created by rule **r16**. Such an auxiliary server has a handler for connecting to the network node and to the saturated server node, and a port for incoming messages. We call a server with an associated server for equilibrating the load, an optimized server. Then, when the network has a message to dispatch to an optimized server, if the number of incoming messages $in(k)$ on the optimized server is smaller that the incoming messages $in(k.0)$ on its associated server, then the message goes to k (rule **r17**), else it goes to $k.0$ (rule **r18**). Since an auxiliary server is created due to an overload of incoming messages, it is obvious that the next message(s) from the network node will be dispatched to the auxiliary node; hence rule **r18** will be executed before rule **r17**, which is expressed by the strategy $\texttt{first}(\mathbf{r18}, \mathbf{r17})$. A message is dispatched from $k.0$ to k (rule **r19**) when $in(k) < in(k.0)$. If an optimised server fails, then the rule **r20** creates a temporary server similarly to rule **r7** which in addition recovers all messages on the auxiliary server which it deletes.

7. Embedding Runtime Verification

We have shown in Sect. 6 how a particular autonomic system can be modeled using the ρ_{pg}-calculus. The model should also ensure formally that the intended self-managing specification of the system helps indeed preserving the properties of the system. Some properties can be verified by checking the presence of particular port graphs. Such properties can be easily encoded as concrete port graphs, abstractions, or strategies, hence as entities of the calculus. Consequently, the properties can be placed at the same level as the specification of the modeled system and they can be tested at any time.

An invariant of the system can be expressed as a port graph rewrite rule with identical sides, $G \Rightarrow G$, testing the presence of a port graph G. The failure of the invariant is handled by a failure port graph STK that does not allow the execution to continue. The strategy verifying such an invariant is then:

$$\texttt{first}(G \Rightarrow G, X \Rightarrow \mathsf{STK})!$$

Such strategy is useful for instance to ensure, in our running example, the persistency of a given critical server of the network, or may be used also to check that there is always a minimal number of servers available in the network. From another perspective, we express the unwanted occurrence of a concrete port graph G in the system using the strategy:

$$(G \Rightarrow \mathsf{STK})!$$

In practice, such strategies are employed in model checking applications to test unwanted situations.

In both cases above, instead of yielding the failure STK signalling that a property of the system is not satisfied, the problem can be "repaired" by associating to each property the necessary rules or strategies to be inserted in the system in case of failure. Such ideas need to be further explored

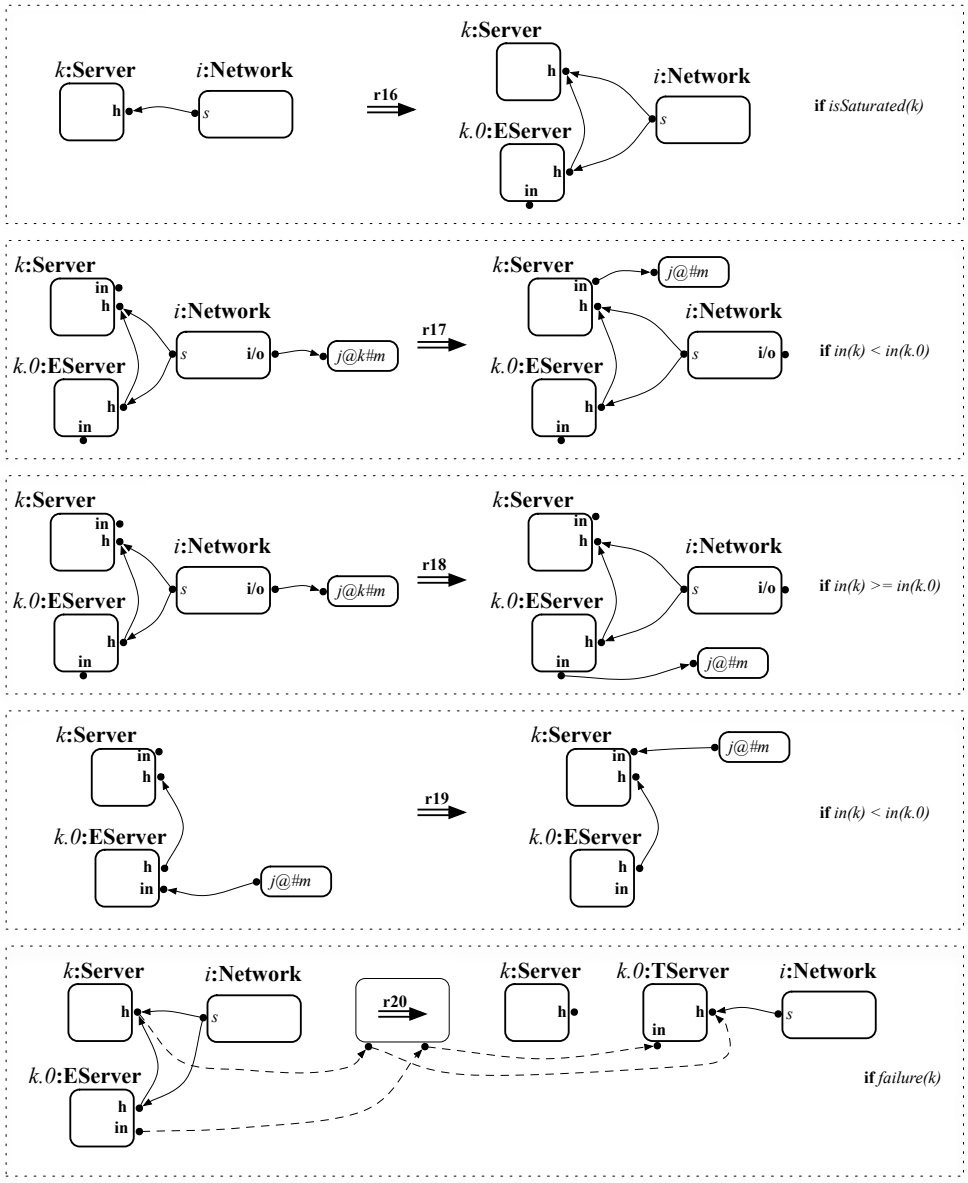

Figure 8. Rules for self-optimisation in the mail delivery system

but open a wide field of possibilities for combining runtime verification and self-healing in ρ_{pg}-calculus.

8. Conclusion

In this paper our main objective was to propose a formalism, the port graph calculus introduced in Section 5, for modeling autonomic systems.

From the computational point of view, we have shown that this calculus allows us to model concurrent interactions between port graph rewrite rules and concrete port graphs, as well as interactions between rewrite rules or interactions creating new rewrite rules. Thanks to strategies, some interactions may be designed with more control. The suitable balance between controlled and uncontrolled interactions is an interesting question to address for a given application. Here again, biological systems may provide us with valuable intuitions.

From the verification point of view, we can take advantage of the classical techniques used in rewriting for checking properties of autonomic systems. For instance, confluence or critical pairs computations help detecting conflicts

in a system evolution. Also some processes may be required to terminate when they are involved in computations. On the contrary, for known non-terminating processes, detecting periodicity of the processes may be of interest. Therefore, further work requires to address the verification of such properties for port graph rewriting. We have also outlined in this paper some ideas for runtime verification of properties in such systems, that need further exploration.

References

[1] O. Andrei and H. Kirchner. A Rewriting Calculus for Multigraphs with Ports. In *Proc. of RULE'07*, 2007.

[2] O. Andrei and H. Kirchner. Graph Rewriting and Strategies for Modeling Biochemical Networks. In *Proc. of NCA'07, a SYNASC Workshop, Romania*, 2007.

[3] E. Balland, P. Brauner, R. Kopetz, P.-E. Moreau, and A. Reilles. Tom: Piggybacking rewriting on java. In *RTA*, volume 4533 of *LNCS*, pages 36–47. Springer, 2007.

[4] E. Balland, P. Brauner, R. Kopetz, P.-E. Moreau, and A. Reilles. Tom Manual. LORIA, Nancy (France), version 2.5, July 2007.

[5] J.-P. Banâtre, P. Fradet, and Y. Radenac. A Generalized Higher-Order Chemical Computation Model. *ENTCS*, 135(3):3–13, 2006.

[6] J.-P. Banâtre, P. Fradet, and Y. Radenac. Programming Self-Organizing Systems with the Higher-Order Chemical Language. *International Journal of Unconventional Computing*, 3(3):161–177, 2007.

[7] J.-P. Banatre and D. Le Metayer. A new computational model and its discipline of programming. Technical Report RR-566, INRIA, 1986.

[8] J.-P. Banâtre, Y. Radenac, and P. Fradet. Chemical Specification of Autonomic Systems. In *IASSE*, pages 72–79. ISCA, 2004.

[9] C. Bertolissi, P. Baldan, H. Cirstea, and C. Kirchner. A Rewriting Calculus for Cyclic Higher-order Term Graphs. *ENTCS*, 127(5):21–41, 2005.

[10] P. Borovanský, C. Kirchner, H. Kirchner, and C. Ringeissen. Rewriting with strategies in ELAN: a functional semantics. *International Journal of Foundations of Computer Science*, 12(1):69–98, February 2001.

[11] L. Cardelli. Brane Calculi. In V. Danos and V. Schächter, editors, *CMSB*, volume 3082 of *LNCS*, pages 257–278. Springer, 2005.

[12] N. Chabrier-Rivier, F. Fages, and S. Soliman. The Biochemical Abstract Machine BIOCHAM. In V. Danos and V. Schächter, editors, *CMSB*, volume 3082 of *LNCS*, pages 172–191. Springer, 2005.

[13] A. J. Chakravarti, G. Baumgartner, and M. Lauria. Application-Specific Scheduling for the Organic Grid. In R. Buyya, editor, *GRID*, pages 146–155. IEEE Computer Society, 2004.

[14] H. Cirstea and C. Kirchner. The rewriting calculus - Part I and II. *Logic Journal of the IGPL*, 9(3):427—498, 2001.

[15] H. Cirstea, C. Kirchner, L. Liquori, and B. Wack. Rewrite strategies in the rewriting calculus. *ENTCS*, 86(4):593–624, Jun 2003.

[16] A. Corradini, U. Montanari, F. Rossi, H. Ehrig, R. Heckel, and M. Löwe. Algebraic Approaches to Graph Transformation - Part I: Basic Concepts and Double Pushout Approach. In G. Rozenberg, editor, *Handbook of Graph Grammars and Computing by Graph Transformations, Volume 1: Foundations*, pages 163–246. World Scientific, 1997.

[17] V. Danos and S. Pradalier. Projective Brane Calculus. In V. Danos and V. Schächter, editors, *CMSB*, volume 3082 of *LNCS*, pages 134–148. Springer, 2004.

[18] A. Habel, R. Heckel, and G. Taentzer. Graph Grammars with Negative Application Conditions. *Fundam. Inform.*, 26(3/4):287–313, 1996.

[19] J. O. Kephart and D. M. Chess. The Vision of Autonomic Computing. *IEEE Computer*, 36(1):41–50, 2003.

[20] C. Kirchner, F. Kirchner, and H. Kirchner. Strategic computations and deductions. To appear, 2007.

[21] C. Kirchner, H. Kirchner, and M. Vittek. Designing Constraint Logic Programming Languages using Computational Systems. In P. Van Hentenryck and V. Saraswat, editors, *Principles and Practice of Constraint Programming. The Newport Papers.*, chapter 8, pages 131–158. MIT Press, 1995.

[22] C. Kirchner, R. Kopetz, and P.-E. Moreau. Anti-Pattern Matching. In *ESOP*, volume 4421 of *LNCS*, pages 110–124. Springer, 2007.

[23] J. W. Klop. Term Rewriting Systems. In S. Abramsky, D. Gabbay, and T. Maibaum, editors, *Handbook of Logic in Computer Science*, volume 1, chapter 6. Oxford University Press, 1990.

[24] J. W. Klop, V. van Oostrom, and F. van Raamsdonk. Reduction strategies and acyclicity. In H. Comon-Lundh, C. Kirchner, and H. Kirchner, editors, *Rewriting, Computation and Proof*, volume 4600. Springer, jun 2007.

[25] C. Laneve and F. Tarissan. A simple calculus for proteins and cells. *ENTCS*, 171(2):139–154, 2007.

[26] N. Martí-Oliet, J. Meseguer, and A. Verdejo. Towards a strategy language for Maude. In N. Martí-Oliet, editor, *WRLA 2004*, volume 117 of *ENTCS*, pages 417–441. Elsevier, 2005.

[27] R. Milner. Pure bigraphs: Structure and dynamics. *Inf. Comput.*, 204(1):60–122, 2006.

[28] D. Plump. Confluence of Graph Transformation Revisited. In A. Middeldorp, V. van Oostrom, F. van Raamsdonk, and R. C. de Vrijer, editors, *Processes, Terms and Cycles: Steps on the Road to Infinity, Essays Dedicated to Jan Willem Klop, on the Occasion of His 60th Birthday*, volume 3838 of *LNCS*, pages 280–308. Springer, 2005.

[29] A. Regev, E. M. Panina, W. Silverman, L. Cardelli, and E. Y. Shapiro. BioAmbients: an abstraction for biological compartments. *TCS*, 325(1):141–167, 2004.

[30] Terese. *Term Rewriting Systems*. Cambridge University Press, 2003. M. Bezem, J. W. Klop and R. de Vrijer, eds.

[31] E. Visser. Stratego: A Language for Program Transformation based on Rewriting Strategies. System Description of Stratego 0.5. In A. Middeldorp, editor, *RTA*, volume 2051 of *LNCS*, pages 357–361. Springer-Verlag, May 2001.